DTIC FILE COPY

AD-A222 499

UMIACS-TR-90-52
CS-TR -2451
April 1990

Using Program Adaptation Techniques for Ada Coverage Testing

James M. Purtilo
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

Elizabeth L. White
Department of Computer Science
University of Maryland
College Park, MD 20742

COMPUTER SCIENCE TECHNICAL REPORT SERIES

DTIC
ELECTE
JUN 07 1990
S E D

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

90 05 21 055

UMIACS-TR-90-52 April 1990
CS-TR -2451

Using Program Adaptation Techniques for Ada Coverage Testing

James M. Purtilo
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

Elizabeth L. White
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Software development processes often employ program metrics tools to analyze software. Examples include static program analysis, code profiles, and coverage analysis. Automated tools can facilitate this study by collecting data and generating the desired measures. Unfortunately, the implementation of such tools has proven to be difficult and time consuming.

In order to simplify the task of constructing analysis tools for Ada, we have examined a new parser generator system for use in a source to source transformation approach. Our goal is to allow developers to easily and compactly express what information they need from the input language in order to compute the desired statistics. One result of our research is AINT, an instrumenter and tester for Ada source code which was constructed using the new translation technology. AINT provides both statement and branch coverage testing of Ada programs, allowing users to evaluate the adequacy of their test data. This paper summarizes the new parsing system (called NewYacc) used to build AINT, then describes the implementation of our testing system.

Research was begun while supported by Office of Naval Research contract N00014-87-K-0307. It is now supported by Honeywell and the DARPA/ISTO Common Prototyping Language initiative, with oversight provided by Office of Naval Research.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

1 OVERVIEW

Software metrics are important to understanding the software development process. In order to employ such techniques, the software being studied must either be analyzed manually, or some automatic tool must be used to process the programs. While small programs may be reasonably processed by hand, this is too time consuming for analysis of larger software systems. Automated tools are necessary, but historically these tools have proven very complex to implement.

The complexity in development of automated tools is the result of several factors. When the metric requires static analysis of programs, the implementor must have access to parsing and other intermediate program representations which are usually not available for general use from a vendor. Vendors market a custom analysis system and users are not able to take this software and modify it for their purposes. Because of this, tools developed for static analysis must be built from scratch or assembled from other previously built pieces.

Other metrics require the more dynamic approach of installing code into the programs to perform the metric computation. Previously, implementors have constructed tools for profiling, coverage, and other metrics by using a low-level approach for adding code as it is compiled. In order to build these tools, the developer needs to know not only about parsing and symbol information, but aliasing, preserving coherent data representation, and code generation, as well.

Many of the static and dynamic metrics can be discussed in terms of the language structure as defined by grammars. This suggests an alternative to the low-level approach traditionally used for program analysis. The idea is to adapt source programs at a high level so that existing compilers, code generators and other processing tools can be employed without alteration. In order to develop automatic tools for use at this high level, a technology which integrates the parsing process with the computation of metrics is needed. Users need the ability to do high-level source to source translations quickly and easily by specifying what needs to be done in terms of the language constructs themselves.

Our current research involves such a translation tool called NewYacc. NewYacc allows users to associate actions and parse tree traversals with the grammar rules for a language. Use of this tool is simple and straightforward. Starting with a grammar for the language being analyzed, actions consisting of outputs, function calls, and tree traversals are added. The resulting parser does source to source translations, either computing the desired metrics directly, or inserting source code for later dynamic measurement.

DTIC COPY INSPECTED 1

In order to evaluate this approach, our work so far has been on two different analysis tools, one static and the other dynamic. ASAPP is an Ada Static Analyzer Program which computes several different metrics including Halstead and cyclic complexity. ASAPP works by traversing the parse tree for the input program and making calls at different points in the tree to do the necessary analysis. The second tool, AINT, is the focus of this paper. AINT is an Ada branch and statement coverage tool. Given an input program, AINT modifies the program in a way that allows a tester to run this output program multiple times on different data sets, and thereby

Codes
Avail and/or
Dist Special
A-1

STATEMENT "A" per Dr. A Van Tilbourg
ONR/Code 1133
TELECON 6/6/90 VG

track which statements and branches were executed on each data set. By knowing that their test data has executed all of the code in certain ways, implementors can have greater confidence that design and implementation errors in the software have been exposed, resulting in increased program quality.

In the next section we introduce the NewYacc development tool, including simple examples. Next, an overview of the AINT testing system is given along with an example of how we augment Ada programs. Finally, we give a short description of how AINT was developed using NewYacc.

2 NEWYACC

Our approach to solving the problems with the generation of automatic tools to operate on complex languages is to use a translation mechanism which allows high-level source to source translation. An example of this kind of mechanism is NewYacc, an extension of the Unix compiler generator tool Yacc [John79]. Yacc takes as input a set of grammar rules with associated actions and produces a parser for the language defined by the input grammar. The parser generated by YACC parses an input stream bottom-up, discarding the parse tree as it reduces by the rules of the grammar. The actions specified by the user are performed as the reductions occur. When the parsing is done, the parse program terminates.

In NewYacc, a parser is also produced from an input set of grammar rules and actions; however, in this parser the parse tree is retained after an input stream is processed. When the sentence has been accepted, the NewYacc system allows the tree to be traversed and additional actions to be performed in user-defined ways. The actions are rewrite rules associated with the productions in the language.

A rule translation in NewYacc is of the form:

```
[(display_mask) translation_body]
```

where the display_mask is a list of display masks and the translation body is a list of grammar symbols, strings, statements, and user-defined functions. A traversal in NewYacc is a user-specified dynamic walk through the parse tree and corresponds to a single translation of the input. The display masks and grammar symbols control the path of the traversal. At each internal node, the translation items are evaluated in order and at each leaf node, the contents are simply output.

For example, a grammar for expressions using only + and − as operators can be written in infix notation as follows:

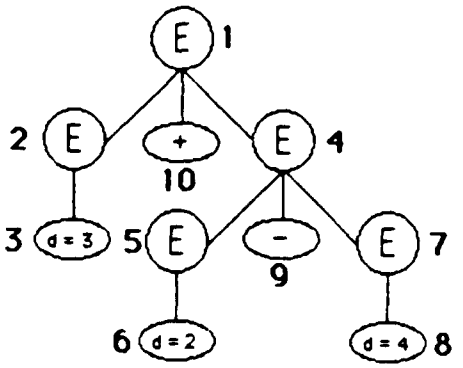

Figure 1: Parse Tree for $3 + 2 - 4$.

```
Expression ::=  Expression '+' Expression  |
                Expression '-' Expression  |
                digit
;
```

A parse tree for the input sentence "$3 + 2 - 4$" is shown in Figure 1. If the task to be performed is the transformation of infix expressions to either postfix expressions or prefix expressions, this could be specified in NewYacc with the following actions:

```
Expression ::=  Expression '+' Expression
        [(POSTFIX)  #1 " " #3 " " #2]
        [(PREFIX)  #2 " " #1 " " #3 ]
   |            Expression '-' Expression
        [(POSTFIX) #1 " " #3 " " #2]
        [(PREFIX) #2 " " #1 " " #3 ]
      |           digit
;
```

The "#" indicates the traversal of the specified subtree, where the subtrees correspond to the different elements on the right hand side of the grammar rule and are numbered from left to right starting with 1. In the above example, the "#1" in the first NewYacc action specifies that the subtree for the first **Expression** is to be traversed. Literal strings such as " " are output when encountered in the translation body. Above, the first NewYacc action for the POSTFIX traversal indicates a POSTFIX traversal of the first subtree, output of the literal string " ", traversal of the third subtree as POSTFIX, output of the string " ", and finally traversal of the second subtree. If the traversal was desired was postorder then the nodes of the parse tree of Figure 1 would be visited in the order indicated by the numbering and the output would be "3 2 4 - +" simply by specifying a POSTFIX traversal.

The above example shows an open traversal on the parse tree; other features of NewYacc include

conditional traversals, use of scope variables, and selective traversals in which leaf nodes are not output. For a more complete language description, see [PuCa89].

3 AINT

The purpose of this section is to describe the AINT tool itself, including a concrete example to illustrate its application. As mentioned earlier, the development of AINT demonstrates how easily individual users can devise their own desired transformations on a source language using the NewYacc system. However, details concerning just *how* NewYacc can generate an AINT tool will be covered in a later section.

Statement coverage is the computation of which statements in the input program were actually executed during a test run. Branch coverage involves examination of all places where control flow can be divided, and informing the tester which execution paths were not followed. This information is necessary for developers to judge the adequacy of their testing data: if they find their test data has caused all parts of the application to be exercised, then they would presumably have increased confidence that any potential faults have been realized; in turn, the more faults that can be recognized through testing, the greater is the likelihood that program errors will be exposed.

In program instrumentation, code is added to an application program so that each time a significant event occurs during a run of the program, the desired event is recorded for later analysis. Any program testing system of this type has several responsibilities: Statically, it must provide a language processor to install this instrumentation code; at run time it must record the execution of statements and branches; and after execution it must analyze the execution trace and inform the user about statement and branch coverage.

The AINT system to instrument code fulfills all of these responsibilities. Each Ada package to be instrumented is input to a tool that outputs a new version of the code (suitably augmented to containing instrumentation calls), a pretty-printed version of the source, and the list of testing obligations to be fulfilled. Subsequently, the obligation lists from all the Ada packages are concatenated and run through a tool which creates a new Ada package (containing the obligations tables for statement and branch coverage); this package of data structures is compiled and linked in with the instrumented packages, along with a driver module. The result is an executable program that not only functions as the original application, but also keeps track of all testing obligations as they are fulfilled. A diagram of the AINT system architecture is in Figure 2.

Once the test program has been created, it can be run any number of times to test sets of data. For example, if the Towers of Hanoi program shown in Figure 3 was executed with an input greater than 0, all of the statements in the main program and in procedure `hanoi` would be executed. On an input of 0, none of the code inside the conditional statement in procedure `hanoi` would be executed although the rest of the code would execute. When AINT is used to

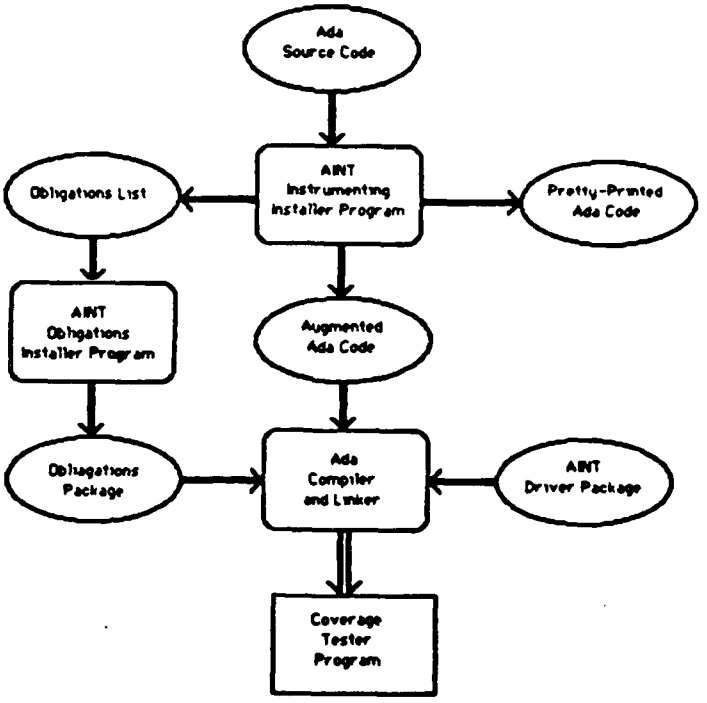

Figure 2: Architecture of AINT.

```
with Text_io; use Text_io;
procedure TOWERS_OF_HANOI is
   package IO_INTEGER is new INTEGER_IO ( INTEGER); use IO_INTEGER;
   NUM_OF_DISKS. NATURAL;
   procedure HANOI (N: NATURAL; X, Y, Z: CHARACTER) is
    begin
       if N /=  0 then
          HANOI(N-1, X, Z, Y);
          PUT("MOVE DISK");
          PUT(n); PUT("from");
          PUT(X); PUT(" to ");
          PUT(y); NEW_LINE;
          HANOI(N-1, Z, Y, X);
      end if;
   end HANOI;

begin
    PUT( "How many disks?"); NEW_LINE;
    GET( NUM_OF_DISKS);
    HANOI(NUM_OF_DISKS, 'X', 'Y', 'Z');
end TOWERS_OF_HANOI;
```

Figure 3: Towers of Hanoi.

instrument the Towers of Hanoi, this the new version can be compiled and run, producing the output shown in Figure 4.

In the first run, all of the statements and branches were executed, but in the second run, the input 0 causes the true branch of the conditional and all of the enclosed statements to be output as uncovered after the run. The statement numbers in the output correspond the the statement locations in a pretty printed version of Hanoi.

The input source code needs to be augmented in such a way that the output version is functionally equivalent to the original code except for the added code to compute coverage. To do this, coverage information needs to be computed before the execution of every line. The initial version of AINT worked by placing a call to the instrumenting package before every line of input code. These calls served to tell the package what statement was going to be executed next. For example, Figure 5 shows the augmented code for a recursive solution to the Towers of Hanoi program. The `Profile` and `ProfileIF` calls are to the instrumenting package `Probe` which keeps track of what statements and branches have been covered. The numbers in the calls indicate the number of the next statement to be executed and the strings indicate in what package the statement is contained.

The functionality of the input Ada code was not modified by AINT; however, there is an additional overhead of one procedure call per statement plus one call per conditional statement which causes a large slowdown in execution. Upon testing the software, this slowdown was determined to be

RUN 1:

```
How many disks?  2
MOVE DISK 1 from X to Y
MOVE DISK 2 from X to Z
MOVE DISK 1 from Y to Z
All statements covered.
All branches covered.
```

RUN 2:

```
How many disks? 0
Statement 11 not covered in "hanoi"
Statement 12 not covered in "hanoi"
Statement 13 not covered in "hanoi"
Statement 14 not covered in "hanoi"
Statement 15 not covered in "hanoi"
Statement 16 not covered in "hanoi"
Statement 17 not covered in "hanoi"
Statement 18 not covered in "hanoi"
Statement 19 not covered in "hanoi"
True branch of statement 10 not covered in "hanoi"
```

Figure 4: Sample output from execution of example program.

```
with Probe; use Probe;
with Text_io; use Text_io;
procedure TOWERS_OF_HANOI is
   package IO_INTEGER is new INTEGER_IO ( INTEGER);
   use IO_INTEGER;
   NUM_OF_DISKS: NATURAL;
   procedure HANOI (N: NATURAL; X, Y, Z: CHARACTER) is
    begin
       Profile(10,"hanoi");
       if N /=  0 then
          ProfileIF(1,10,"hanoi");
          Profile(11,"hanoi"); HANOI(N-1, X, Z, Y);
          Profile(12,"hanoi"); PUT("MOVE DISK");
          Profile(13,"hanoi"); PUT(N);
          Profile(14,"hanoi"); PUT("from");
          Profile(15,"hanoi"); PUT(X);
          Profile(16,"hanoi"); PUT(" to ");
          Profile(17,"hanoi"); PUT(Y);
          Profile(18,"hanoi"); NEW_LINE;
          Profile(19,"hanoi"); HANOI(N-1, Z, Y, X);
      ELSE
          ProfileIF(0,10,"hanoi");
      end if;
   end HANOI;

begin
    Profile(24,"hanoi"); PUT("How many disks?");
    Profile(25,"hanoi"); NEW_LINE;
    Profile(26,"hanoi"); GET(NUM_OF_DISKS);
    Profile(27,"hanoi"); HANOI(NUM_OF_DISKS, 'X', 'Y', 'Z');
end TOWERS_OF_HANOI;
```

Figure 5: Augmented version of Towers of Hanoi.

```
with Probe; use Probe;
with Text_io; use Text_io;
procedure TOWERS_OF_HANOI is
   package IO_INTEGER is new INTEGER_IO ( INTEGER);
   use IO_INTEGER;
   NUM_OF_DISKS: NATURAL;
   procedure HANOI (N: NATURAL; X, Y, Z: CHARACTER) is
   begin
       AINTstatement(AINThanoi,10) := true;
       if N /=  0 then
        AINTbranch(AINThanoi,10,1) := true;
        AINTstatement(AINThanoi,11) := true; HANOI(N - 1, X, Z, Y);
        AINTstatement(AINThanoi,12) := true; PUT("MOVE DISK");
        AINTstatement(AINThanoi,13) := true; PUT(N);
        AINTstatement(AINThanoi,14) := true; PUT("from");
        AINTstatement(AINThanoi,15) := true; PUT(X);
        AINTstatement(AINThanoi,16) := true; PUT(" to ");
        AINTstatement(AINThanoi,17) := true; PUT(Y);
        AINTstatement(AINThanoi,18) := true; NEW_LINE;
        AINTstatement(AINThanoi,19) := true; HANOI(N - 1, Z, Y, X);
      ELSE
        AINTbranch(AINThanoi,10,0):= true;
      end if;
   end HANOI;

begin
    AINTstatement(AINThanoi,24) := true; PUT("How many disks?");
    AINTstatement(AINThanoi,25) := true; NEW_LINE;
    AINTstatement(AINThanoi,26) := true; GET(NUM_OF_DISKS);
    AINTstatement(AINThanoi,27) := true; HANOI(NUM_OF_DISKS, 'X', Y', 'Z');
end TOWERS_OF_HANOI;
```

Figure 6: Second augmented version of Towers of Hanoi.

excessive and a new approach was tried. Instead of placing procedure calls before every line of code, it was decided to put the instrumenting code there and avoid making any procedure calls not in the original code. In order to do this, the data structures of the instrumenting package had to be changed. Multidimensional arrays were used to keep track of obligation table information and to allow fast computation of what obligations had been covered. Figure 6 shows what the newer version of AINT will output for the Towers of Hanoi problem. `AINTstatement` and `AINTbranch` are boolean arrays in which the individual elements are set to true when the appropriate statement is encountered. The first array is two-dimensional with the first dimension indicating in which package the statement occurs and the second dimension indicating the statement number. The second array contains a third dimension which indicates which branch of the statement has been encountered. Initialization of these arrays is done in the `Probe` package which is created by the obligations installer. At the end of a test run, these arrays are examined to determine what obligations were not fulfilled.

Using this new approach, the instrumenting overhead is less substantial: one constant-time array assignment per statement plus one constant-time array assignment per conditional. Although both versions added the same number of lines of code to the input program, the execution time of a single array assignment takes *much* less time than the execution of a procedure call along with the corresponding procedure code. Overhead for the first attempt at AINT increased exponentially with the number of statements executed; however, for most code, overhead for the second version is less than one hundred percent. Code which has many conditional statements and few or no procedure calls has slightly higher overhead.

While these figures sound poor at first, the second AINT implementation represents a design compromise. Remember that our objective is for developers to find *easy* ways to tailor instrumentation tools. The sub- one hundred percent figure is a fairly reasonable cost given that an execution event must be recorded for each program statement in order to obtain true statement coverage. It is reasonable to ask whether the statement execution obligations could be recorded on a *block*, not individual statement, basis, which is in fact how many other internal testing system are implemented. Unfortunately, when we considered how to perform block-level instrumentation, we discovered the grammar we were working with would have required our writing a much larger number of elaborate statements (to deal with such situations as handling of `GO TO` statements, and so on), all while being careful not to accidentally introduce unreachable code (e.g., by adding an obligations check after a `RETURN` statement.) The statement level instrumentation, on the other hand, was much simpler in design, and provided what was to us an acceptable testing performance.

Overall, the development of the AINT system took approximately 50 hours, from inception to demonstration of a distributable product to our funding agent. Most of this time was spent on the initial version of the translator, used to augment application programs. The remaining time was spent on extending the AINT system as described earlier, (informally) validating that our translator would perform reliably on a local test suite of Ada programs (used to ensure we provided suitable translations for code involving all Ada language constructs)[1], and finally checking the costs of addition instrumentation to a large application provided by our industrial funding agent, for use at their site. The DEC VMS -based implementation of our AINT system, running on a Microvax, added instrumentation code to an 18.000 source line Ada program in a matter of a few minutes.

4 IMPLEMENTATION OF AINT

In order to perform statement and branch coverage for input source code, an instrumenter needs to know what statements and branches must be executed and which are actually executed during a run. The way AINT accomplishes this is by building an obligations table and then marking

[1] We regret to report that our grammar, which is independent of NewYacc, still fails to correctly parse Ada code involving static initialization of multi-dimensional arrays.

statements and branches as they are executed in a test run. Upon completion of the run, any unmarked obligations are output.

Development of AINT was a three step process. The first step was to add the NewYacc actions to an Ada grammar so that the necessary statements and branches would be added to the obligations table and the input source code would be augmented with the appropriate instrumenting statements. For each Ada statement type, three tasks were added as actions for the grammar rule:

1. a call is made to a procedure which enters the statement into the obligations table.
2. instrumenting code is added to the output.
3. the Ada statement is output by traversing the statement itself.

For example, the Ada grammar rule and NewYacc specification for an assignment statement is as follows:

```
sim_statement : assignment_stmt
[(AINT) probe() #1 bump()]
```

The `probe()` call adds the statement to the obligations table and outputs the appropriate instrumenting statement. The #1 indicates a traversal of the assignment statement itself, and the `bump()` increments a global statement counter.

Branch coverage required a little more creativity. To compute coverage for IF statements, two instrumenting statements were added per input IF statement, one for each possible branch. There are eight steps for the coverage of IF statements:

1. a call is made to a procedure which enters the true and false branch of the IF statement into the obligations table.
2. the **IF condition THEN** part of the statement is output.
3. the instrumenting code for the true branch is output.
4. the statements for the true branch are traversed and output (along with their associated statement instrumenting procedure calls).
5. the **ELSE** is output.
6. the instrumenting code for the false branch is output.
7. the statements for the false branch are traversed.
8. the **END IF** is output.

The following is a simplified version of how the above steps could be specified in NewYacc for an IF statement.

```
conditional  : IF condition THEN stmtlist ELSE stmtlist END IF
[(AINT) #1 #2 #3 probeif("then") #4 #5 probeif("else") #6 #7 #8]
```

The probeif() calls add the obligations to the table and output the appropriate code. The "#" symbols indicate traversal of the appropriate parts of the statement.

More complex augmentation must be done for IF statements without an associated ELSE branch, and for IF statements that use the ELSIF construction. In the first case, an ELSE branch which has only the single instrumenting statement is added in the augmented code. ELSIF requires more work, as the code is rewritten as cascaded IF- THEN-ELSE statements to simplify the computation of the branch coverage. Fig. 7(a) shows a small Ada program which contains both of these IF statement types. The augmented code (with statement instrumenting code omitted) for branch coverage is shown in Fig. 7(b).

The second development step involved creating a tool which would take the obligation lists as input and output the necessary obligation tables. To do this, a grammar written to parse the obligation lists was augmented with NewYacc statements in such a way that the entire package was generated by the tool. Figure 8 shows the data structures package which would be produced for the Towers of Hanoi program in Figure 6. This package initializes array elements for all of the statements and branches to be covered as false and all other statements and branches as true. The enumerated type **AINTtestnames** contains one element for each input package being covered and the constant **AINTdimensionlimit** gives the largest statement number in any of the input packages. In order to build this package, first the obligations installer outputs the header information, then it traverses the input obligations file three times. The first traversal outputs the declarations for **AINTtestnames**, **AINTdimensionlimit** and the array type declarations. Then the obligation list is traversed twice; first to find all of the statements which must be executed in the packages and finally to find the branches in the packages. The **end probe;** is output at the end of the installer run.

As the final step in the development of AINT, it was necessary to write an Ada driver package that would call the Ada source code being tested, and then inform the tester of the results of the run. Computing the results of the run proved to be very simple, given the data structures used by the instrumenting process. Any element of the two instrumenting arrays which is false at completion time, signals an unfulfilled obligation.

5 CONCLUSION

The study of the software development process requires numerous tools to facilitate analysis. Automated tools are desirable for this study. Traditionally, this instrumentation and analysis

```
with Text_io; use Text_io;
procedure If_statements(in X: NATURAL)  is
begin
   IF x = 0 THEN
        x := x + 1;
   END IF;
   IF x < 3 THEN
        x := x + 3;
   ELSEIF x < 5 THEN
        x := x + 5;
   ELSE
        x := x + 1;
   END IF;
end If_statements;
```

(a)

```
with Text_io; use Text_io;
procedure If_statements(in X: NATURAL)  is
begin
   IF x = 0 THEN
      AINTbranch(AINTif_statements,5,1) := true; x := x + 1;
   ELSE
      AINTbranch(AINTif_statements,5,0) := true;
   END IF;
   IF x < 3 THEN
      AINTbranch(AINTif_statements,8,1) := true; x := x + 3;
   ELSE
      AINTbranch(AINTif_statements,8,0) := true;
      IF x < 5 THEN
            AINTbranch(AINTif_statements,10,1) := true; x := x + 5;
      ELSE
            AINTbranch(AINTif_statements,10,0) := true; x := x + 1;
      END IF;
   END IF;
end If_statements;
```

(b)

Figure 7: Branch Coverage Augmentation.

```
package probe is
AINTdimensionlimit : constant integer := 27;
Type AINTtestnames is (AINThanoi);
Type AINTstmtarray is array(AINTtestnames'First..
                    AINTtestnames'Last,1..AINTdimensionlimit) of boolean;
Type AINTbrcharray is array(AINTtestnames'First..
                    AINTtestnames'Last,1..AINTdimensionlimit, 0..1) of boolean;
AINTstatement: AINTstmtarray :=
                    AINTstmtarray'(AINThanoi => (10 => false, 11 => false, 12 =>
                    false, 13 => false, 14 => false, 15 => false, 16 => false,
                    17 => false, 18 => false, 19 => false, 24 => false, 25 =>
                    false, 26 => false, 27 => false, others => true));
AINTbranch: AINTbrcharray :=
                    AINTbrcharray'(AINThanoi=> (10 =>(others => false), others =>
                    (others => true)));
end probe;
```

Figure 8: Obligations Package.

has been performed *internally* by the language processors and execution environment. But this has made development of such tools time consuming, and, even worse, has made it difficult for ordinary users to change or adapt such tools for their own needs.

We have shown how *external* instrumentation and analysis can be performed with relative ease by ordinary users. Construction of the necessary language processors and execution environment is made easier by availability of translation tools such as NewYacc. The user may specify a desired source to source translation in terms of the constructs of the language being examined. These specifications are added to a grammar to create a parser in which the parse tree for input sentences can be traversed in ways specific to the application. Our experiences have been that this adaptation can be performed with acceptable costs in terms of execution time.

Our claim concerning the utility of external adaptation has been supported by our presentation of AINT, an instrumentation and testing system for Ada code. AINT does simple branch and statement coverage analysis by performing a high-level source to source translation of the input code into a version of the code which, when used with the AINT system packages, does the computations necessary for the analysis. The development of this simple testing system AINT demonstrates the ease with which other developers can build tools for different applications, including testing, debugging, and code profiling.

REFERENCES

[Gilb77] Gilb, T., **Software Metrics**, Winthrop, 1977.

[HoMi81] Howden, W.E. and E. Miller, eds. **Tutorial: Software Testing and Validation Techniques**, IEEE, (1981).

[Huan78] Huang, J.C. Program Instrumentation and Software Testing, **Computer**, vol. 11(4), (April 1978), pp. 25-31.

[John79] Johnson, S. YACC: Yet Another Compiler Compiler. Bell Laboratories, (1979).

[Myer79] Myers, G.J., **The Art of Software Testing**, John Wiley & Sons, New York, 1979.

[PuCa89] Purtilo, J., and J. Callahan. Parse Tree Annotations. **Communications of the ACM**, (December 1989).

[Watt87] Watt, D. et al. **Ada Language and Methodology.**, Prentice-Hall, (1987).